Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Факультет вычислительной математики и информатики
Кафедра экономико-математических методов и статистики

РАБОТА ПРОВЕРЕНА
Рецензент,

_____

« »_____2015 г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой, д. ф.-м. н., профессор
_____ А.В. Панюков

« »_____2015 г.

Параллельная реализация метода эллипсоидов для задач оптимизации большой размерности

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ ЮУрГУ-010400.62.2015.001.001 ВКР

Консультант,
_____

« »_____2015 г.

Руководитель проекта,
_____ В.А. Голодов

« »_____2015 г.

Автор проекта
студент группы ВМИ-413
_____ В.А. Безбородов
« »_____2015 г.

Нормоконтролер, к. ф.-м. н., доцент
_____ Т.А. Макаровских
« »_____2015 г.

Челябинск, 2015

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Факультет вычислительной математики и информатики
Кафедра экономико-математических методов и статистики

УТВЕРЖДАЮ
Заведующий кафедрой, д. ф.-м. н., профессор
_____ А.В. Панюков
« » _____2015 г.

З А Д А Н И Е

на выпускную квалификационную работу студента

Безбородова Вячеслава Александровича

Группа ВМИ-413

1. Тема работы: Параллельная реализация метода эллипсоидов для задач оптимизации большой размерности.
   Утверждена приказом по университету от « » _____2015 г.
   №_____
2. Срок сдачи студентом законченной работы « » _____2015 г.
3. Исходные данные к работе
   3.1. Данные из учебной литературы;
   3.2. Самостоятельно сконструированные тестовые данные.
4. Перечень вопросов, подлежащих разработке
   4.1. Изучение общей схемы работы метода эллипсоидов;
   4.2. Изучение приемов параллельной обработки данных;
   4.3. Разработка класса (типа данных) для реализации параллельно выполняемых операций над матрицами с применением библиотеки GMP;
   4.4. Разработка параллельной реализации метода эллипсоидов для задачи линейного программирования;

4.5. Оценка сложности полученной реализации;

4.6. Сравнение с известными методами решения;

4.7. Тестирование;

4.8. Проверка на модельных данных.

5. Перечень графического материала

6. Календарный план

Наименование этапов дипломной работы	Срок выполнения этапов работы	Отметка о выполнении
1. Сбор материалов и литературы по теме дипломной работы	02.02.2015 г.	
2. Исследование способов построения математической модели задачи		
3. Разработка математической модели и алгоритма		
4. Реализация разработанных алгоритмов		
5. Проведение вычислительного эксперимента		
6. Подготовка пояснительной записки дипломной работы		
Написание главы 1		
Написание главы 2		
Написание главы 3		
7. Оформление пояснительной записки		
8. Получение отзыва руководителя		
9. Проверка работы руководителем, исправление замечаний		
10. Подготовка графического материала и доклада		
11. Нормоконтроль		
12. Рецензирование, представление зав. кафедрой	10.06.2015 г.	

7. Дата выдачи задания « »_____2015 г.

Заведующий кафедрой_____/А.В. Панюков /

Руководитель работы_____/В.А. Голодов /

Студент_____/В.А. Безбородов /

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
«Южно-Уральский государственный университет»
(Национальный исследовательский университет)
Факультет вычислительной математики и информатики
Кафедра экономико-математических методов и статистики

АННОТАЦИЯ

Безбородов, В.А. Параллельная реализация метода эллипсоидов для задач оптимизации большой размерности / В.А. Безбородов. – Челябинск: ЮУрГУ, Факультет вычислительной математики и информатики, 2015. – 59 с., 7 ил., 3 табл., 2 прил., библиогр. список – 0 названий.

В дипломной работе произведен анализ алгоритма метода эллипсоидов на предмет вычислительной сложности выполняемых операций. На основе результатов анализа разработана параллельная реализация метода эллипсоидов, адаптированная для решения задач оптимизации большой размерности на многопроцессорных и/или многоядерных вычислительных системах с общей разделяемой памятью.

Приведены результаты вычислительных экспериментов, продемонстрирован пример решения задачи оптимизации большой размерности с применением разработанной программной реализации.

ОГЛАВЛЕНИЕ

Введение

Задачи оптимизации получили чрезвычайно широкое распространение в технике, экономике, управлении. Типичными областями применения теории оптимизации являются прогнозирование, планирование промышленного производства, управление материальными ресурсами, а также контроль качества выпускаемой продукции [**?**].

Успешность хозяйственной деятельности зависит от того, как распределяются имеющиеся ограниченные ресурсы. В связи с тем, что такая задача оптимального распределения довольно часто возникает на практике в различных сферах жизнедеятельности, актуальным становится поиск способов ускорения ее решения.

Задачи оптимизации большой размерности характеризуются высокой трудоемкостью. Использование доступного ресурса аппаратного параллелизма современных вычислительных систем рассматривается как возможность ускорения поиска их решения. Применение библиотек, реализующих поддержку арифметики произвольной точности, диктуется необходимостью достижения высокой точности при решении практических задач.

Разрабатывали, а впоследствии развивали метод эллипсоидов такие ученые, как Шор Н.З. [**?**], Юдин Д.Б., Немировский А.С. [**?**], Хачиян Л.Г. [**?**], Гершович В.И. [**?**], Стецюк П.И. [**?**] и др.

В работе исследован алгоритм метода эллипсоидов. Разработана его программная реализация, ориентированная на многопроцессорные и/или многоядерные вычислительные системы с общей разделяемой памятью. Показано приложение программной реализации к решению задачи оптимизации большой размерности.

Целями работы являются:

1) разработка параллельной реализации метода эллипсоидов, поддерживающей арифметику произвольной точности;
2) использование полученной реализации метода эллипсоидов для решения задачи оптимизации большой размерности.

В соответствии с поставленными целями в работе решаются следующие **задачи**:

- Исследование операций классического алгоритма метода эллипсоидов на вычислительную сложность;
- Разработка программной реализации алгоритма с распараллеливанием наиболее длительных по времени операций;
- Обеспечение поддержки арифметики расширенной и произвольной точности;
- Проверка и тестирование разработанного программного обеспечения.

Объектом исследования данной работы является метод эллипсоидов, **предметом** – параллельная реализация метода, поддерживающая арифметику произвольной точности.

Работа состоит из введения, 2 глав, заключения и списка литературы. Объем работы составляет 59 страниц. Список литературы содержит 0 наименований.

В первой главе рассматривается алгоритм метода эллипсоидов, производится его анализ на предмет вычислительной сложности с целью поиска наиболее ресурсоемких операций, нуждающихся в ускорении путем распараллеливания.

Во второй главе описывается способ параллельной реализации метода эллипсоидов, аргументированно доказывается необходимость обеспечения поддержки арифметики расширенной точности, а также описывается тестовое окружение разрабатываемого класса.

В заключении перечислены основные результаты работы.

1 Метод эллипсоидов

В настоящее время большое внимание уделяется созданию автоматизированных систем планирования, проектирования и управления в различных областях промышленности. На первый план выдвигаются вопросы качества принимаемых решений, в связи с чем возрастает роль методов и алгоритмов решения оптимизационных задач в математическом обеспечении автоматизированных систем различного уровня и назначения.

Имеется несколько основных источников, порождающих задачи оптимизации: задачи математического программирования, задачи нелинейного программирования, задачи оптимального управления, задачи дискретного программирования или задачи смешанного дискретно-непрерывного типа [**?**].

Сфера применения методов оптимизации огромна. Создание эффективных методов оптимизации является ключом к решению многих вычислительных проблем математического программирования, особенно для задач большой размерности.

Для минимизации гладких функций широко применяются различные модификации градиентных процессов, поскольку направление антиградиента в данной точке локально является направлением наискорейшего спуска. Регулировка шага в большинстве алгоритмов этого типа основана на том, чтобы обеспечить монотонное и в достаточной степени «существенное» уменьшение значения функции на каждом шаге.

Простейший обобщенный градиентный метод состоит в движении на каждом шаге в направлении, обратном направлению обобщенного градиента. Этот метод под названием обобщенного градиентного спуска (ОГС) предложен Н.З. Шором в 1961 г. в связи с необходимостью разработки эффективного алгоритма решения транспортных задач большой размерности для задач текущего планирования, решаемых в Институте кибернетики АН УССР совместно с Госпланом УССР. Впервые метод обобщенного градиентного спуска для минимизации кусочно-линейных выпуклых функций использовался при решении транспортных и транспортно-производственных задач [**?**]. Затем метод ОГС был распространен на класс произвольных выпуклых функций [**?**]

и на задачи выпуклого программирования в гильбертовом пространстве [?]. Широкое распространение получили стохастические аналоги ОГС [?]. Алгоритмы решения задач математического программирования, построенные на основе ОГС, отличаются простотой и, что особенно важно для задач большой размерности, экономным использованием оперативной памяти ЭВМ.

К ограничениям метода ОГС относятся его довольно медленная сходимость, сложность контроля точности решения и то, что он применим только к классу выпуклых функций.

В 1969–1970 гг. Н.З. Шором были предложены ускоренные варианты обобщенных градиентных методов, основанные на использовании операции растяжения пространства в направлении градиента и разнсти двух последовательных градиентов [?]. Идея этих методов существенно отлична от той, которая используется для ускорения сходимости в случае гладких функций – идеи квадратичной аппроксимации функции в окрестности минимума, в той или иной мере определяющей формализм как методов сопряженных градиентов, так и квазиньютоновских методов [?]. В то же время предельные варианты методов с растяжением пространства при определенных условиях регулярности и гладкости обладают свойством квадратичной скорости сходимости. Таким образом, предложенные алгоритмы обладают высокой эффективностью и применительно к гладким задачам минимизации. В дальнейшем алгоритмы с растяжением пространства были обобщены на задачи нахождения локальных минимумов невыпуклых негладких функций [?].

В США и Западной Европе градиентными методами минимизации негладких функций всерьез начали заниматься примерно с 1973 г. сначала в связи с приложениями в области дискретного программирования [?], а затем в целом для решения задач большой размерности [?]. Результаты работ в этом направлении на Западе достаточно полно представлены в сборнике [?]. Особенно интенсивно развивается направление так называемой ε-субградиентной оптимизации, по идее близкое, с одной стороны, к алгоритмам В.Ф. Демьянова решения минимаксных задач, а с другой, особенно в формальном отношении, – к алгоритмам метода сопряженных градиентов (или «давидоновского» типа).

И наконец, в последнее время обнаружились очень интересные связи между алгоритмами последовательных отсечений и алгоритмами с растяжением пространства [**?**, **?**].

Таким образом, область обобщенных градиентных методов оптимизации не представляет нечто окончательно сформировавшееся и застывшее, а, наоборот, быстро развивается.

1.1 Алгоритм метода эллипсоидов

Рассмотрим алгоритм решения задачи выпуклого программирования, гарантирующий уменьшение объема области, в которой локализуется оптимум, со скоростью геометрической прогрессии, причем знаменатель этой прогрессии зависит только от размерности задачи. Этот алгоритм относится к классу алгоритмов обобщенного градиентного спуска с растяжением пространства в направлении градиента (ОГСРП) [**?**].

Пусть имеется задача выпуклого программирования:

$$\min f_0(x) \tag{1.1}$$

при ограничениях

$$f_i(x) \leq 0, \quad i = 1, \ldots, m, \quad x \in E_n, \tag{1.2}$$

где E_n – евклидово пространство размерности n, $f_\nu(x)$, $\nu = \overline{0, m}$ – выпуклые функции, определенные на E_n; $g_\nu(x)$ – субградиенты соответствующих функций. Пусть имеется априорная информация о том, что существует оптимальная точка $x^* \in E_n$ (не обязательно единственная), которая находится в шаре радиуса R с центром в точке x_0 (формально к системе ограничений 1.2 можно добавить ограничение $||x - x_0|| \leq R$).

Рассмотрим следующий итеративный алгоритм (при $n > 1$).

Определение 1.1. Оператором растяжения пространства E_n в направлении ξ с коэффициентом β называется оператор $R_\beta(\xi)$, действующий на вектор x

Алгоритм 1 Метод эллипсоидов

Шаг 0. Инициализация.
Положить $x_k = x_0$; $B_k = E$, где E – единичная матрица размерности $n \times n$; $h_k = \frac{R}{n+1}$ – коэффициент, отвечающий за уменьшение объема шара. Перейти к шагу 1.
Шаг 1. Вычислить

$$g(x_k) = \begin{cases} g_0(x_k), & \text{если } \max\limits_{1 \le i \le m} f_i(x_k) \le 0, \\ g_{i^*}(x_k), & \text{если } \max\limits_{1 \le i \le m} f_i(x_k) = f_{i^*}(x_k) > 0. \end{cases}$$

Если $g(x_k) = 0$, то завершить алгоритм; x_k – оптимальная точка. Иначе перейти к шагу 2.
Шаг 2. Вычислить $\xi_k = \dfrac{B_k^T g(x_k)}{||B_k^T g(x_k)||}$. Перейти к шагу 3.
Шаг 3. Вычислить $x_{k+1} = x_k - h_k \cdot B_k \cdot \xi_k$. Перейти к шагу 4.
Шаг 4. Вычислить $B_{k+1} = B_k \cdot R_\beta(\xi_k)$, где $R_\beta(\xi_k)$ – оператор растяжения пространства в направлении ξ_k с коэффициентом β (см. определение 1.1), $\beta = \sqrt{\frac{n-1}{n+1}}$. Перейти к шагу 5.
Шаг 5. Вычислить $h_{k+1} = h_k \cdot r$, где $r = \frac{n}{\sqrt{n^2-1}}$. Перейти к шагу 1.

следующим образом [**?**]:

$$R_\beta(\xi)x = \left(E + (\beta - 1)\xi_k \xi_k^T\right) x.$$

Рассмотрим вопрос оценки скорости сходимости метода эллипсоидов. Покажем, что данный вариант алгоритма ОГСРП сходится по функционалу со скоростью геометрической прогрессии, причем знаменатель этой прогрессии зависит только от размерности задачи.

Лемма 1.1. Последовательность $\{x_k\}_{k=0}^{\infty}$, генерируемая алгоритмом 1, удовлетворяет неравенству

$$||A_k(x_k - x^*)|| \le h_k \cdot (n+1), \quad A_k = B_k^{-1}, \quad k = 0, 1, 2, \ldots \tag{1.3}$$

Доказательство леммы для краткости изложения опущено и может быть найдено в [**?**].

Множество точек x, удовлетворяющих неравенству

$$||A_k(x_k - x)|| \leq (n+1)h_k = R \cdot \left(\frac{n}{\sqrt{n^2-1}}\right)^k,$$

представляет собой эллипсоид Φ_k, объем которого $v(\Phi_k)$ равен

$$\frac{v_0 R^n \left(\frac{n}{\sqrt{n^2-1}}\right)^{nk}}{\det A_k},$$

где v_0 – объем единичного n-мерного шара. Получаем

$$\frac{v(\Phi_{k+1})}{(\Phi_k)} = \frac{\left(\frac{n}{\sqrt{n^2-1}}\right)^n \cdot \det A_k}{\det A_{k+1}} = \frac{\left(\frac{n}{\sqrt{n^2-1}}\right)^n \cdot \det A_k}{\det R_\alpha(\xi_k) \cdot \det A_k} = \frac{1}{\alpha}\left(\frac{n}{\sqrt{n^2-1}}\right)^n =$$

$$= \sqrt{\frac{n-1}{n+1}}\left(\frac{n}{\sqrt{n^2-1}}\right)^n = q_n < 1.$$

Таким образом, объем эллипсоида, в котором локализуется оптимальная точка x^* в соответствии с неравенством (1.3), убывает со скоростью геометрической прогрессии со знаменателем q_n.

1.2 Вычислительная сложность операций метода эллипсоидов

В индустрии разработки программного обеспечения известен феномен, который состоит в том, что на 20% методов программы приходится 80% времени ее выполнения [**?**]. Такое эмпирическое правило хорошо согласуется с более общим принципом, известным как принцип Парето либо «правило 80/20».

Утверждение 1 (Принцип Парето). 80% результата можно получить, приложив 20% усилий.

Относящийся не только к программированию, этот принцип очень точно характеризует оптимизацию программ [**?**]. В работе [**?**] Дональд Кнут указал, что менее 4% кода обычно соответствуют более чем 50% времени выполнения

программы. Опираясь на принцип Парето, можно сформулировать последовательность действий, приводящих к ускорению работы имеющихся программ: необходимо найти в коде «горячие точки» и сосредоточиться на оптимизации наиболее трудоемких процессов.

Проанализируем подробнее вычислительную сложность операций алгоритма метода эллипсоидов. Для этого введем некоторые обозначения из области асимптотического анализа [**?**].

Определение 1.2. Функция $f(n)$ *ограничена сверху* функцией $g(n)$ асимптотически с точностью до постоянного множителя, т.е.

$$f(n) = O\left(g(n)\right),$$

если существуют целые N и K, такие, что $|f(n)| \leq Kg(n)$ при всех $n \geq N$.

Определение 1.3. Функция $f(n)$ *ограничена снизу* функцией $g(n)$ асимптотически с точностью до постоянного множителя, т.е.

$$f(n) = \Omega\left(g(n)\right),$$

если существуют целые N и K, такие, что $f(n) \geq Kg(n)$ при всех $n \geq N$.

Определение 1.4. Функция $f(n)$ *ограничена снизу и сверху* функцией $g(n)$ асимптотически с точностью до постоянного множителя, т.е.

$$f(n) = \Theta\left(g(n)\right),$$

если одновременно выполнены условия определений 1.2 и 1.3.

На **шаге 1** алгоритма 1 изменение текущего субградиента $g(x_k)$ происходит на основании анализа значений функций ограничений (1.2) в текущей точке x_k, т.е. анализируется последовательность значений

$$\max_{1 \leq i \leq m} f_i(x_k).$$

Вычислительная сложность поиска максимального из m чисел зависит от выбора алгоритма.

При использовании линейного последовательного поиска цикл выполнит m итераций. Трудоемкость каждой итерации не зависит от количества элементов, поэтому имеет сложность $T^{iter} = O(1)$. В связи с этим, верхняя оценка всего алгоритма поиска $T_m^{min} = O(m) \cdot O(1) = O(m \cdot 1) = O(m)$. Аналогично вычисляется нижняя оценка сложности, а в силу того, что она совпадает с верхней, можно утверждать $T_m^{min} = \Theta(m)$.

При использовании алгоритма бинарного поиска на каждом шаге количество рассматриваемых элементов сокращается в 2 раза. Количество элементов, среди которых может находиться искомый, на k-ом шаге определяется формулой $\frac{m}{2^k}$. В худшем случае поиск будет продолжаться, пока в массиве не останется один элемент, т.е. алгоритм имеет логарифмическую сложность: $T_m^{binSearch} = O\left(\log(m)\right)$. Резюмируем все вышесказанное относительно алгоритмов поиска в виде таблицы 1.

Таблица 1 — Оценка сложности некоторых алгоритмов поиска

Алгоритм	**Структура данных**	**Временная сложность**		**Сложность по памяти**
		В среднем	**В худшем**	**В худшем**
Линейный поиск	Массив из n элементов	$O(n)$	$O(n)$	$O(1)$
Бинарный поиск	Отсортированный массив из n элементов	$O\left(\log(n)\right)$	$O\left(\log(n)\right)$	$O(1)$

Однако при использовании некоторых алгоритмов (например, алгоритма бинарного поиска) потребуются дополнительные процедуры для упорядочивания входной последовательности значений. В таблице 2 представлены асимптотические оценки наиболее известных алгоритмов сортировки массива из n элементов[1].

[1]Более подробный анализ приведен в [**?**].

Таблица 2 — Оценка сложности некоторых алгоритмов сортировки[1]

Алгоритм	**Временная сложность**			**Сложность по памяти**
	В лучшем	**В среднем**	**В худшем**	**В худшем**
Быстрая сортировка	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	$O(n)$
Сортировка слиянием	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$
Пирамидальная сортировка	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(1)$
Пузырьковая сортировка	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка вставками	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Сортировка выборомм	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Блочная сортировка	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(nk)$
Поразрядная сортировка	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$

На **шаге 2** алгоритма производится вычисление нормированного обобщенного градиента

$$\xi_k = \frac{B_k^T g(x_k)}{||B_k^T g(x_k)||},$$

что подразумевает выполнение трудоемких матричных операций, таких как транспонирование, умножение на вектор и на число. Асимптотическая сложность таких операций для матрицы размерности $n \times n$ может быть оценена как $O(n^2)$.

На **шаге 3** при обновлении значения текущей точки выполняется умножение матрицы B_k на вектор ξ_k:

$$x_{k+1} = x_k - h_k \cdot B_k \cdot \xi_k.$$

Аналогично предыдущей оценке, цена такой операции составит $O(n^2)$.

Шаг 4 наиболее сложен из-за необходимости вычисления оператора растяжения пространства

$$B_{k+1} = B_k \cdot R_\beta(\xi_k).$$

Если представить оператор в матричной форме (определение 1.1), то можно видеть, что в процессе его вычисления используются все вышеперечисленные операции, а также операция сложения матриц, сложность которой составляет $O(n^2)$.

На **шаге 5** осуществляется пересчет коэффициента h_k, отвечающего за уменьшение объема шара

$$h_{k+1} = h_k \cdot r.$$

Эта операция может быть выполнена за константное время, т.е. асимптотически ее сложность составит $O(1)$.

Из проведенного анализа вычислительной сложности операций, входящих в алгоритм метода эллипсоидов, можно сделать несколько выводов. Во-первых, учитывая специфику рассматриваемого класса задач (задачи оптимизации большой размерности), наиболее трудоемкие операции будут выполняться значительно дольше менее трудоемких, что приведет к сильно неравномерной загрузке вычислительной системы. Во-вторых, схема чередования сложных/простых в вычислительном смысле операций, а также выбор целевой платформы (многопроцессорные и/или многоядерные системы с общей разделяемой памятью) наталкивают на возможность использования Fork-Join Model (FJM) модели распараллеливания задач для ускорения работы алгоритма метода эллипсоидов.

На основании полученных данных можно сформулировать гипотезу о том, насколько удастся ускорить выполнение метода в целом для решения задач оптимизации большой размерности, если к наиболее ресурсоемким операциям применить алгоритм распараллеливания по данным.

Гипотеза 1 (О соотношении времени). Пусть $f(t)$ – это время работы алгоритма метода эллипсоидов для задачи оптимизации размерности $N \times M$, выполняемого *в однопоточном режиме*. Тогда для параллельной реализации метода эллипсоидов, выполняемой *в многопоточном режиме*, для достаточно больших N и M справедливо равенство

$$F(t) = kf(t),$$

где $F(t)$ – общее время работы параллельной реализации метода, а k – коэффициент ускорения $(k > 1)$.

Приближенная оценка для коэффициента ускорения k может быть получена в ходе выполнения вычислительных экспериментов.

Выводы по главе один

Показано, что широта и разнообразие применения методов оптимизации обуславливают необходимость создания эффективных методов оптимизации для решения различных вычислительных проблем математического программирования.

Доказано, что метод эллипсоидов гарантированно локализует оптимум со скоростью геометрической прогрессии при условии существования оптимальной точки.

Проведенный анализ метода эллипсоидов показал, что некоторые операции метода вычислительно сложны.

Выдвинута гипотеза о том, что на современных многопроцессорных и/или многоядерных системах с общей разделяемой памятью можно добиться ускорения работы метода эллипсоидов, если распараллелить наиболее ресурсоемкие операции.

2 Параллельная реализация метода эллипсоидов

В процессе решения любой задачи оптимизации возникает необходимость оперирования над матрицами, которые составляются исходя из условий конкретной задачи. Поскольку в данной работе изначально заложена ориентация на решение задач оптимизации большой размерности, то и матрицы, возникающие из анализа условий этих задач, будут иметь большую размерность. Любое численное решение должно достигать определенной точности результата. А при оперировании матрицами большой размерности обеспечение требуемой точности становится еще более актуальным.

Один из способов обеспечения поддержки арифметики расширенной точности заключается в использовании готовых специализированных математических библиотек, таких, например, как GMP.

2.1 Парадигма Fork-Join

В параллельном программировании, Fork-Join model (модель ветвление-объединение, FJM) – это способ запуска и выполнения параллельных участков кода, при котором выполнение ветвей завершается в специально обозначенном месте для того, чтобы в следующей точке продолжить последовательное выполнение. Параллельные участки могут разветвляться рекурсивно до тех пор, пока не будет достигнута заданная степень гранулярности задачи. Модель была впервые сформулирована в 1963 г., и может рассматриваться как один из параллельных паттернов проектирования [**?**].

Различные реализации FJM обычно управляют *задачами*, *волокнами* или *легковесными нитями*, а не *процессами* уровня операционной системы, и используют *пул потоков* для их выполнения. Специальные ключевые слова позволяют программисту определять точки *возможного параллелизма*; проблема создания реальных потоков и управления ими ложится на реализацию.

В [**?**] приведена иллюстрация парадигмы Fork-Join. Представим ее на рисунке 1. Здесь три участка программы потенциально разрешают параллельное исполнение различных блоков. Последовательное выполнение показано сверху, в то время как его Fork-Join эквивалент снизу.

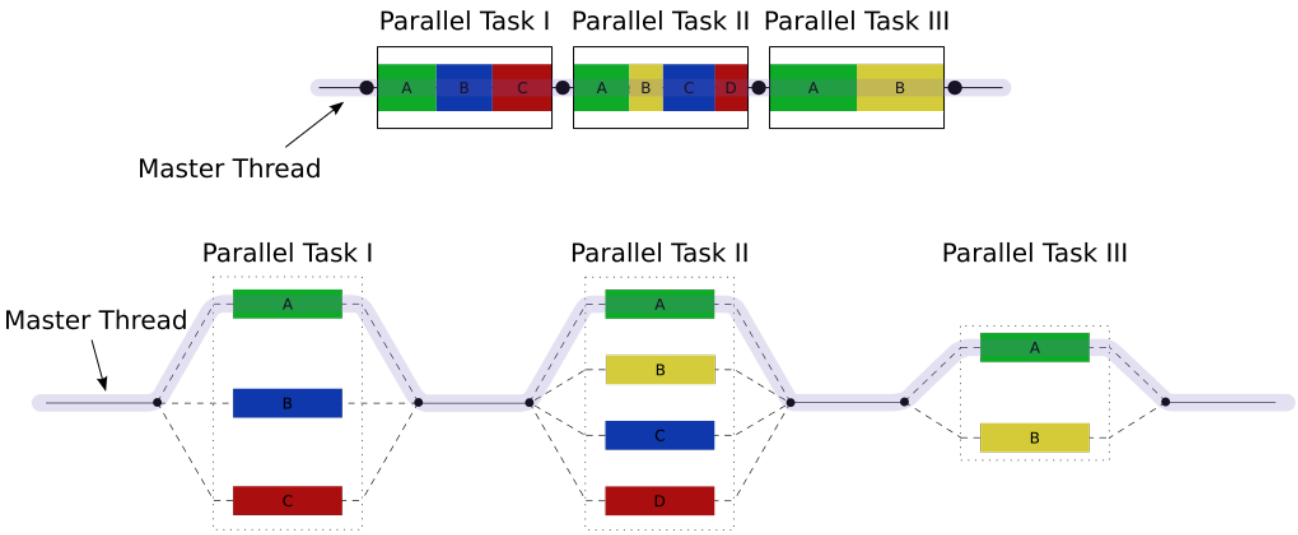

Рисунок 1 — Иллюстрация парадигмы Fork-Join

Легковесные нити, используемые в Fork-Join программировании, обычно имеют свой собственный планировщик, который управляет ими, применяя схему пула потоков. Такой планировщик может быть гораздо проще полнофункционального планировщика задач, применяемого операционной системой. Планировщики потоков общего пользования обязаны приостанавливать/запускать потоки в обозначенных программистом местах, в то время как в парадигме Fork-Join потоки блокируются только в местах непосредственного слияния.

Модель Fork-Join – основная модель параллельного исполнения в технологии OpenMP. Также эта модель поддерживается в Java concurrency framework, в Task Parallel Library для .NET и в Intel Threading Building Blocks. Языки программирования Cilk и Cilk Plus имеют встроенную поддержку FJM в форме ключевых слов *spawn–sync* и *cilk_spawn–cilk_sync* соответственно.

2.2 Необходимость использования библиотеки GMP

Библиотека GMP [?] – *бесплатная (свободная)* библиотека для арифметики произвольной точности, выполняемой над знаковыми целыми, рациональными числами и числами с плавающей запятой. При этом практически не существует предела для точности вычислений, если не считать объем

доступной памяти ЭВМ, на которой производятся вычисления. GMP имеет богатый набор функций, которые имеют стандартизированный интерфейс.

В основном GMP применяется в криптографических, научно-исследовательских приложениях, приложениях, отвечающих за безопасность в сети Интернет, различных системах вычислительной алгебры и т.д.

Основными целевыми платформами GMP являются Unix-подобные системы, такие как GNU/Linux, Solaris, HP-UX, Mac OS X/Darwin, BSD, AIX и проч. Также поддерживается работа на Windows в 32 и 64-битном режимах.

Библиотека GMP тщательно спроектирована для того, чтобы вычисления производились настолько быстро, насколько это возможно одновременно и для больших, и для малых операндов. Такая скорость возможна благодаря использованию машинных слов в качестве базового арифметического типа и быстрых алгоритмов, включающих высокооптимизированный ассемблерный код для большинства внутренних циклов для целого набора наиболее популярных современных центральных процессоров.

Первая версия GMP вышла в 1991 году. С тех пор библиотека непрерывно улучшается и поддерживается, обеспечивая выход новых версий примерно раз в год.

Начиная с версии 6, GMP распространяется одновременно под двумя лицензиями: GNU LGPL v3 и GNU GPL v2. Такое лицензирование позволяет использовать библиотеку бесплатно, изменять ее и публиковать результат.

Приведем численный пример, наглядно доказывающий необходимость использования библиотеки GMP для поддержки арифметики расширенной и произвольной точности. Рассмотрим следующий код, представленный в листинге 1.

```
1  MatrixRandomFiller filler;
2  MatrixMultiplier multiplier;
3  MatrixPrinter printer( &std::cout );
4  printer.setPrecision( 20 );
5
6  std::size_t nRows = 2;
7  std::size_t nColumns = 2;
8
9  Matrix< double > matrix_a( nRows, nColumns );
10 Matrix< double > matrix_b( nRows, nColumns );
```

```
11
12    filler.fill( matrix_a );
13    filler.fill( matrix_b );
14
15    printer.print( matrix_a );
16    printer.print( matrix_b );
17
18    Matrix< double > matrix_c( nRows, nColumns );
19    matrix_c = multiplier.multiply( matrix_a, matrix_b );
20
21    Matrix< double > matrix_d( nRows, nColumns );
22    matrix_d = matrix_a * matrix_b;
23
24    printer.print( matrix_c );
25    printer.print( matrix_d );
```

Листинг 1 — Исходный код примера

В листинге используются объекты следующих классов:

1) `Matrix< Type >` представляет в программе матрицу, элементы которой имеют тип `Type`;
2) `MatrixRandomFiller` предназначен для заполнения матрицы случайными значениями, равномерно распределенными на интервале $[0, 1)$ (в процессе заполнения используются стандартный генератор псевдослучайных чисел `std::default_random_engine` и равномерное распределение `std::uniform_real_distribution` $P(i|a, b) = \frac{1}{b-a}$ с параметрами $a = 0$, $b = 1$);
3) `MatrixMultiplier` производит перемножение матриц;
4) `MatrixPrinter` отвечает за печать матриц.

В строках 1–4 создаются и настраиваются следующие вспомогательные объекты: `filler`, `multiplier` и `printer`. `Printer` настраивается таким образом, чтобы осуществлять вывод в стандартный поток вывода `std::cout` и печатать матрицы с точностью до 20 знаков после запятой. В строках 6–7 задается размерность матриц, участвующих в данном примере. Затем происходит создание (строки 9–10), заполнение случайными значениями (строки 12–13) и печать (строки 15–16) двух операндов. В строке 19 осуществляется умножение матриц `matrix_a` и `matrix_b` стандартным базовым ал-

горитмом (строка на столбец), результат которого сохраняется в объекте `matrix_c`. В строке 22 также осуществляется умножение операндов. Отличие от строки 19 состоит в том, что в данном случае для умножения используется функция-член класса `Matrix`. Результат второго умножения сохраняется в переменной `matrix_d`. После этого осуществляется печать полученных результатов (строки 24–25).

Поскольку матрицы заполняются случайным образом, вывод программы зависит от конкретного запуска. В частности, возможен и такой вариант вывода.

$$\texttt{matrix\_a} = \begin{pmatrix} 0.54846871850742517918 & 0.84412510404456542190 \\ 0.95768018538410970564 & 0.80660421324152631328 \end{pmatrix},$$

$$\texttt{matrix\_b} = \begin{pmatrix} 0.96126112219013459814 & 0.79203852615307757112 \\ 0.79959221481843290036 & 0.90154689901948148467 \end{pmatrix},$$

$$\texttt{matrix\_c} = \begin{pmatrix} 1.20217751736546674124 & 1.19542672538356331557 \\ 1.56553517904925\underline{709736} & 1.48571112974158303643 \end{pmatrix},$$

$$\texttt{matrix\_d} = \begin{pmatrix} 1.20217751736546674124 & 1.19542672538356331557 \\ 1.56553517904925\underline{687532} & 1.48571112974158303643 \end{pmatrix}.$$

Из приведенных значений элементов матриц хорошо видно, что для одних и тех же операндов результат перемножения, полученный вне и внутри класса `Matrix`, различен, начиная с 15-го знака после запятой. При этом популярный математический сервис WolframAlpha [**?**] для данных операндов

выдает результат

$$\begin{pmatrix} 1.2021775173654667\underline{500} & 1.1954267253835633\underline{424} \\ 1.5655351790492\underline{570011} & 1.4857111297415\underline{829851} \end{pmatrix},$$

в котором наблюдаются отличия в знаках во всех элементах матрицы. Попытка провести аналогичные вычисления на языке программирования R приводит к результату, показанному в листинге 2.

```
> options(digits=20)
> lhs = matrix( c( 0.54846871850742517918, 0.84412510404456542190,
                   0.95768018538410970564, 0.80660421324152631328 ),
                nrow=2, ncol=2, byrow=TRUE )
> rhs = matrix( c( 0.96126112219013459814, 0.79203852615307757112,
                   0.79959221481843290036, 0.90154689901948148467 ),
                nrow=2, ncol=2, byrow=TRUE )
> lhs %*% rhs
                       [,1]                  [,2]
[1,] 1.2021775173654667412 1.1954267253835633156
[2,] 1.5655351790492570974 1.4857111297415830364
```

Листинг 2 — Пример вычисления на языке R

Такое расхождение неслучайно. В численном эксперименте (см. листинг 1) участвуют матрицы, все элементы которых имеют тип `double`. В языке C++ для представления чисел с плавающей запятой существуют несколько стандартных типов данных [**?**]. Характеристики этих типов приведены в таблице 3.

Таблица 3 — Размер и диапазон чисел с плавающей запятой в языке C++

Тип	**Размер, байт**	**Допустимый диапазон значений**
float	4	+/– 3.4e +/– 38 (точность ~7 цифр)
double	8	+/– 1.7e +/– 308 (точность ~15 цифр)
long double	8	+/– 1.7e +/– 308 (точность ~15 цифр)

Согласно информации из таблицы 3, тип `double` имеет точность, в два раза превышающую `float`. В общем случае `double` имеет 15–16 десятичных знаков точности, в то время как `float` только 7.

В численном эксперименте использовался тип данных `double`, что вызвало ошибку округления примерно на 15-м знаке после запятой. Соответственно, если применять тип данных с меньшей (`float`) или большей (`long double`) точностью, меняться будет только положение десятичного знака, на котором будет начинаться расхождение, но *ошибка при этом не исчезнет*.

В такой ситуации разумным видится использование специализированных библиотек, таких как GMP, поддерживающих более точные вычисления. Именно свободный доступ, минимальные ограничения на использование, а также основная функция – поддержка арифметики произвольной точности – делают библиотеку GMP идеальным инструментом для использования в данной работе, особенность которой состоит в оперировании матрицами больших размерностей, что налагает дополнительные (более строгие) ограничения на точность полученного результата.

2.3 Распараллеливание матричных операций

В первой главе показано, что метод эллипсоидов активно использует матричные операции. Вычислительная сложность матричных операций варьируется от $O(n^2)$ (сложение, вычитание, транспонирование) до $O(n^3)$ (умножение). Ускорив выполнение этих операций, можно добиться ускорения выполнение всего метода.

Стандартная библиотека C++ не предоставляет класса для представления матриц в программе. Для языка существуют различные реализации класса матриц, работающие без использования многопоточности [**?**, **?**, **?**]. Для ускорения работы метода эллипсоидов требуется разработать класс матриц, поддерживающий распараллеливание матричных операций.

Полный исходный код класса приведен в приложении А . Здесь рассматриваются основные идеи, лежащие в данной реализации.

Класс `Matrix` – шаблонный класс, параметризованный типом `T` элементов матрицы.

```
template< typename T >
class Matrix
```

В классе используется следующие обозначения типов.

```
typedef T value_t;
typedef Matrix< value_t > my_t;
typedef std::size_t index_t;
```

Класс предоставляет три конструктора. Первый создает матрицу указанной размерности, все элементы которой заполнены начальным значением.

```
Matrix( index_t rows, index_t columns, const value_t& value = value_t( 0 ) )
```

Второй – конструктор копирования – создает матрицу, все элементы которой равны элементам указанной матрицы.

```
Matrix( const my_t& rhs )
```

Третий – конструктор перемещения – берет во владение ресурсы временно созданной матрицы.

```
Matrix( my_t&& rhs )
```

Для индексного доступа к элементам применяются две версии оператора вызова функции (для константных и неконстантных объектов).

```
value_t& operator() ( const index_t& row, const index_t& column )

const value_t& operator() ( const index_t& row, const index_t& column ) const
```

Имеются две версии оператора присваивания (по копии и по временному объекту).

```
my_t& operator= ( const my_t& rhs )

my_t& operator= ( my_t&& rhs )
```

Для класса перегружены основные бинарные операторы.

```
my_t operator+ ( const my_t& rhs ) const

my_t operator- ( const my_t& rhs ) const

my_t operator* ( const my_t& rhs ) const
```

Операторы сравнения выполняют проверку на равенство элементов двух матриц.

```
bool operator== ( const my_t& rhs ) const

bool operator!= ( const my_t& rhs ) const
```

Для операции транспонирования класс предоставляет функцию.

```
my_t& transpose()
```

Две функции для получения информации о размерности матрицы.

```
std::size_t get_rows() const

std::size_t get_columns() const
```

Это открытый интерфейс класса `Matrix`, доступный пользователю. Помимо перечисленного функционала, класс содержит статическую функцию для доступа к объекту, управляющему механизмом параллельного выполнения операций.

```
static ParallelHandler* getParallelHandler()
```

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данный момент свидетельствует о наличии *параллелизма по данным* при выполнении матричных расчетов и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между потоками. Выбор способа разделения матриц приводит к определению конкретного метода параллельных вычислений; существование разных схем распределения данных порождает целый ряд *параллельных алгоритмов матричных вычислений*.

Наиболее общие и широко используемые способы разделения матриц состоят в разбиении данных на *полосы* (по вертикали или горизонтали) или на прямоугольные фрагменты (*блоки*) [**?**].

При *ленточном* (block–striped) разбиении каждому потоку выделяется то или иное подмножество строк (*rowwise* или *горизонтальное разбиение*) или столбцов (*columnwise* или *вертикальное разбиение*) матрицы (графическая иллюстрация перечисленных подходов представлена на рисунке 2 [**?**]). Разделение строк и столбцов на полосы в большинстве случаев происходит на *непрерывной* (*последовательной*) основе. При таком подходе для горизонтального разбиения по строкам, матрица A представляется в виде

$$A = (A_0, A_1, \ldots, A_{p-1})^T, A_i = (a_{i_0}, a_{i_1}, \ldots, a_{i_{k-1}}), i_j = ik + j, 0 \leq j < k, k = m/p,$$

где $a_i = (a_{i1}, a_{i2}, \ldots, a_{in}), \quad 0 \leq i < m$, есть i-я строка матрицы A (предполагается, что количество строк m кратно числу вычислительных элементов

(процессоров и/или ядер) p, т.е. $m = k \cdot p$). Существуют и другие способы разбиения, например, блочный.

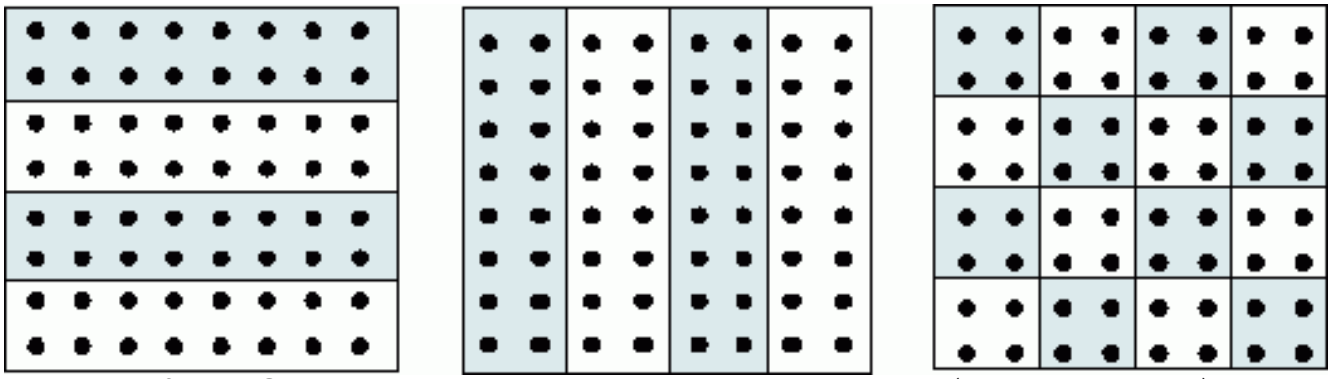

Рисунок 2 — Способы разбиения элементов матрицы (слева направо): горизонтальный, вертикальный и блочный

`ParallelHandler` инкапсулирует параллельное выполнение матричных операций, применяя ленточное разбиение данных на непрерывной основе. Для этого класс предоставляет шаблонную функцию, параметризованную типом итератора и функции.

```
template< typename Iterator, typename Function >
void loop_for( const Iterator& first,
  const Iterator& last,
  Function&& function )
```

Функция создает и управляет потоками, количество которых зависит от текущих настроек. Класс оперирует понятием *политики распараллеливания*, которое может обозначать один из двух типов: *автоматически* и *вручную*.

```
enum ParallelPolicy
{
  policy_automatic,
  policy_direct
};
```

Для установки этих и других значений применяется следующий набор функций.

```
void setAutoParallelPolicy();
void setDirectParallelPolicy( std::size_t numThreads );
void setMinPerThread( std::size_t minPerThread );
void setMinNumThreads( std::size_t minNumThreads );
```

Функции имеют следующие значения (в порядке объявления): использовать автоматический расчет количества потоков, задать количество потоков

вручную, задать минимальную ширину ленты при разбиении данных, установить нижнюю границу для числа используемых потоков.

Для разработанного класса необходимо провести анализ эффективности, доказывающий его преимущество перед последовательным выполнением.

Алгоритмы параллельных матричных операций, основанные на ленточном горизонтальном разбиении матрицы, обладают хорошей «локализацией вычислений», т.е. каждый поток параллельной программы использует только «свои» данные, и ему не требуются данные, которые в данный момент обрабатывает другой поток, нет обмена данными между потоками, не возникает необходимости синхронизации. Это означает, что практически не существуют накладные расходы на организацию параллелизма (за исключением расходов на создание/завершение потоков), и можно ожидать линейного ускорения.

Однако, как видно из представленных ниже результатов, ускорение, которое демонстрируют параллельные матричные операции, далеко от линейного.

Задача умножения матриц обладает сравнительно невысокой вычислительной сложностью – трудоемкость алгоритма имеет порядок $O(n^2)$. Такой же порядок – $O(n^2)$ – имеет и объем данных, обрабатываемый алгоритмом умножения. Время решения задачи одним потоком складывается из времени, когда процессор непосредственно выполняет вычисления, и времени, которое тратится на чтение необходимых для вычислений данных из оперативной памяти в кэш память. При этом время, необходимое для чтения данных, может быть сопоставимо или даже превосходить время счета.

Проведем вычислительный эксперимент: измерим время выполнения последовательного и параллельного алгоритма суммирования и умножения матриц. Столбиковые диаграммы замеров времени выполнения операций сложения и умножения матриц представлены на рисунках 3 и 4 соответственно.

При суммировании элементов матрицы на каждой итерации цикла выполняется простая операция сложения двух чисел. Здесь и далее под *ускорением* выполнения операции будем понимать отношение времени выполнения операции в многопоточном режиме ко времени выполнения той же операции в

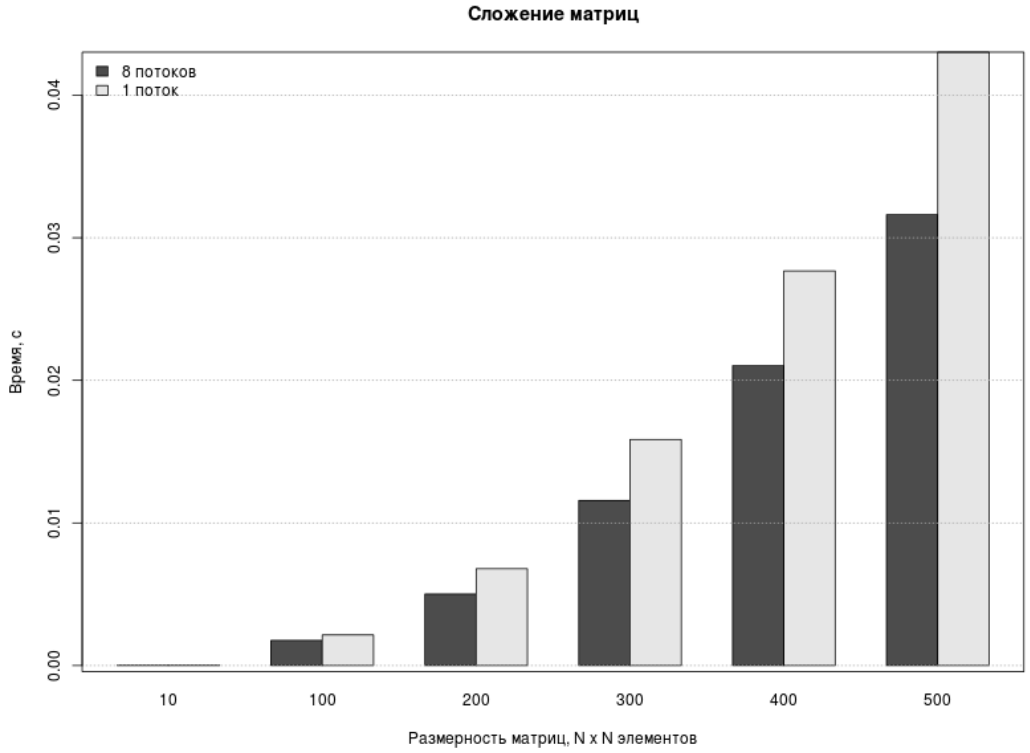

Рисунок 3 — Время выполнения операции сложения матриц

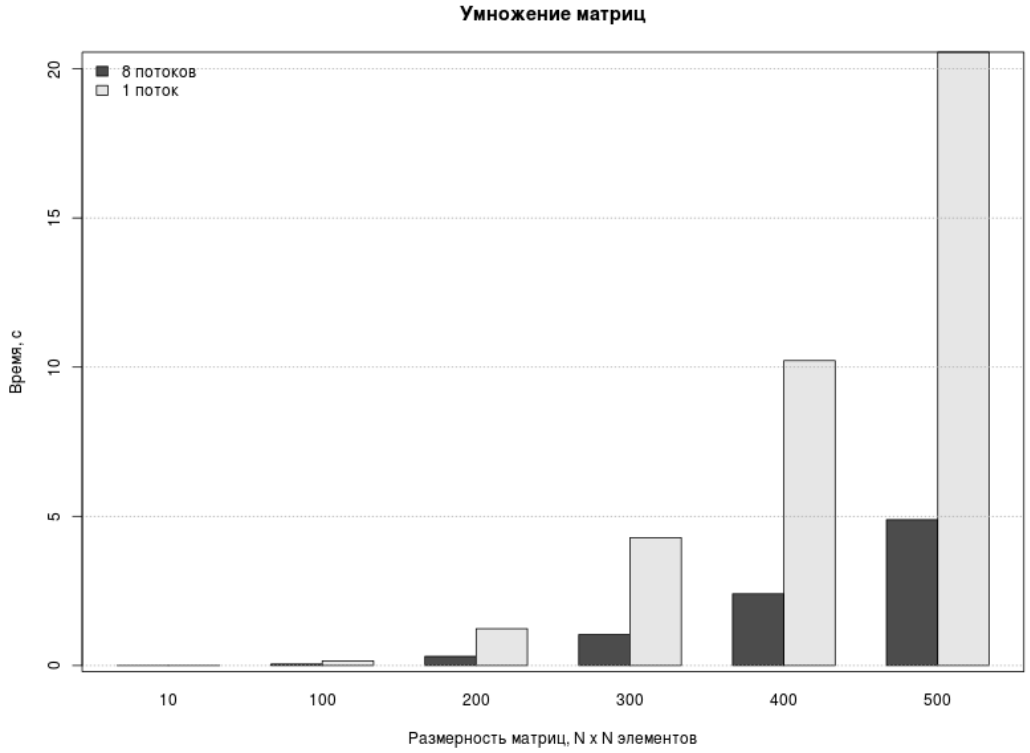

Рисунок 4 — Время выполнения операции умножения матриц

однопоточном режиме. Достигнутое ускорение для операции сложения матриц различных размерностей представлено на рисунке 5.

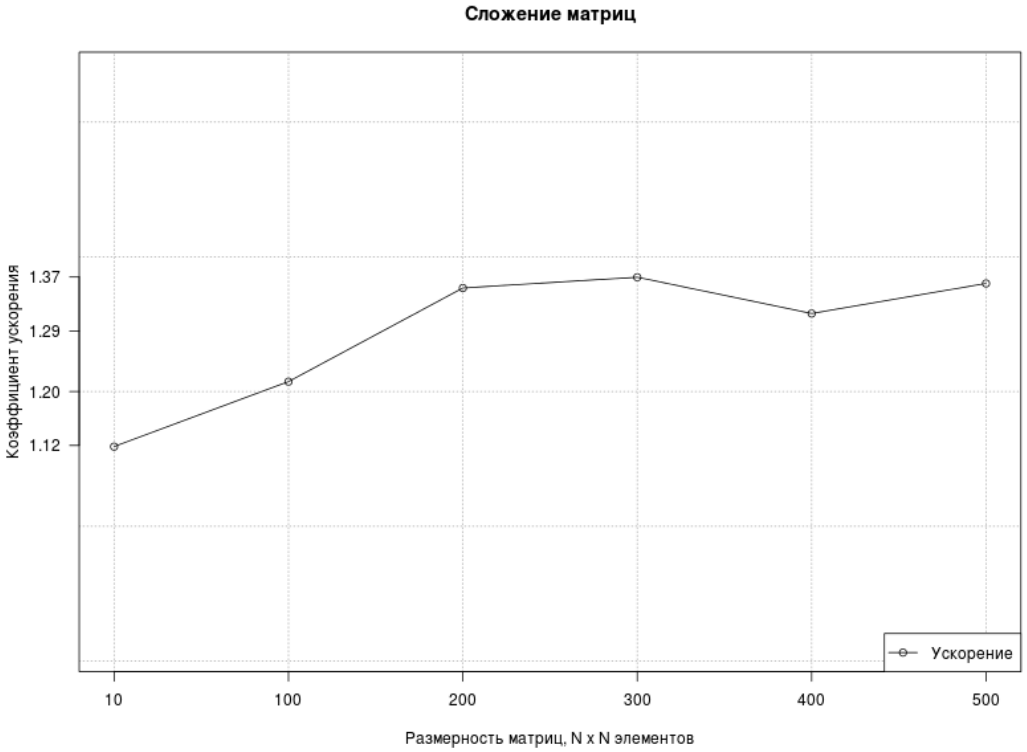

Рисунок 5 — Ускорение операции сложения матриц

Как видно из представленных рисунков, несмотря на меньшую вычислительную сложность, время работы параллельного алгоритма умножения матриц превосходит время выполнения однопоточной версии в среднем всего в 1.3 раза. Этот эксперимент можно рассматривать, как подтверждение предположения о том, что значительная часть времени тратится на выборку необходимых данных из оперативной памяти в кэш процессора.

При умножении матриц на каждой итерации цикла выполняются две операции: более сложная операция умножения и операция сложения. Достигнутое ускорение для операции умножения матриц представлено на рисунке 6.

Проведем другой эксперимент. Зафиксируем и положим равной 500×500 размерность матриц–операндов. Будем выполнять операцию умножения матриц, применяя каждый раз различное количество потоков, чтобы определить поведение функции времени. Результаты эксперимента представлены на рисунке 7.

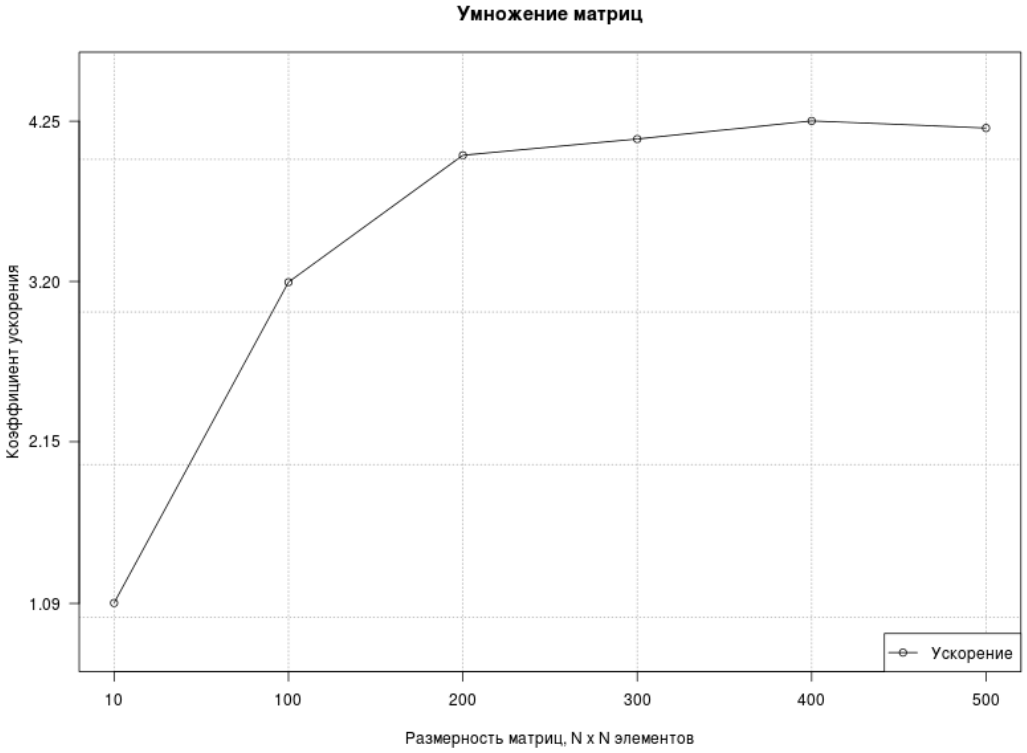

Рисунок 6 — Ускорение операции умножения матриц

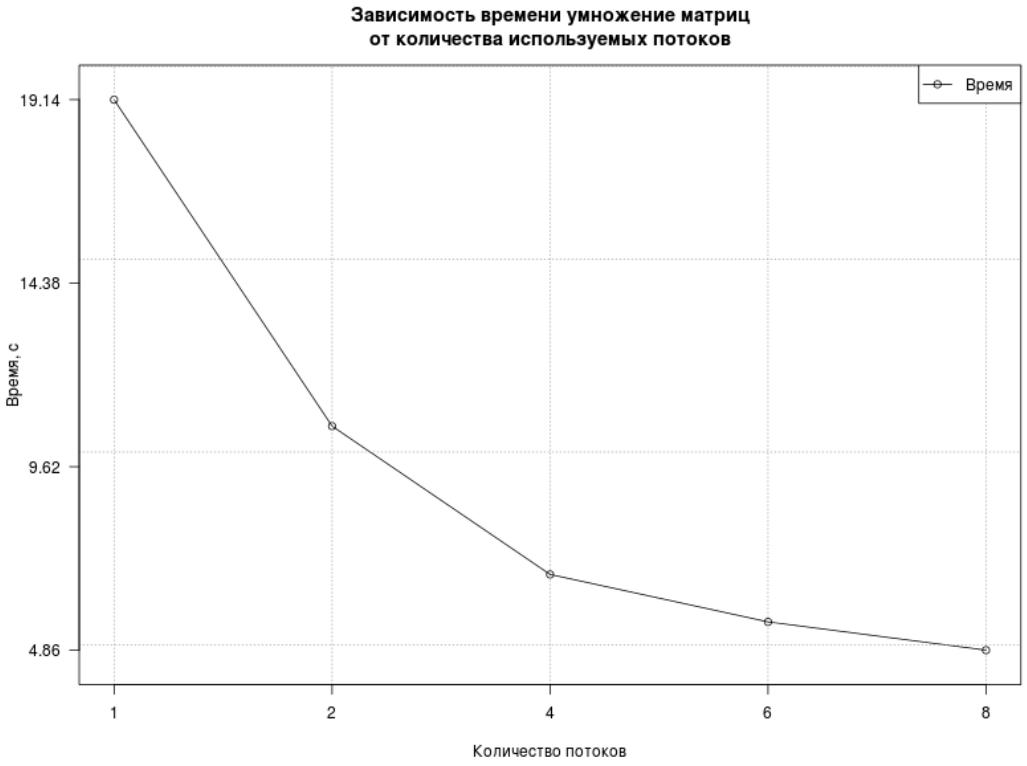

Рисунок 7 — Зависимость времени выполения операции умножения матриц размерности 500×500 от числа используемых потоков

Из рисунка 7 видно, что линейное увеличение количества используемых потоков приводит к нелинейному падению времени выполнения операции умножения. Из анализа результатов эксперимента также следует, что использование числа потоков большего, чем аппаратно поддерживается оборудованием, не приведет к дальнейшему падению функции времени. Такое замедление будет возникать из-за частых переключений планировщика потоков для симуляции параллелизма.

Таким образом, представленная реализация класса, представляющего матрицы в программе, действительно выполняется *быстрее* и имеет *ускорение, большее 1*, благодаря распараллеливанию вычислительно сложных матричных операций.

2.4 Тестирование

Тестирование – важная часть процесса разработки программных продуктов, поскольку ошибки – это не случайное явление.

Различают две формы тестирования [**?**].

1) *Тестирование методом прозрачного ящика* (white box testing), в котором испытатель имеет понятие о внутренней работе программы.
2) *Тестирование с алгоритмом типа черного ящика* (black box testing), когда программа тестируется без использования информации о ее реализации.

Обе формы тестирования имеют важное значение для проектов, претендующих на попадание в категорию высококачественных. Чаще всего используется тестирование по принципу черного ящика, поскольку оно, как правило, моделирует типичное поведение пользователя. Например, при «черном» тестировании компоненты интерфейса можно рассматривать как кнопки. Если испытатель, щелкнув по кнопке, не увидел никакой реакции на свое действие, он делает вывод о том, что, очевидно, в программе есть ошибка.

Тестирование по принципу черного ящика не может охватить все аспекты испытываемого продукта. Современные программы слишком велики, чтобы реализовать имитацию щелчков по каждой кнопке, проверить все возможные варианты входных данных и выполнить все комбинации команд. Необходимость «белого» тестирования объясняется тем, что гораздо проще гарантировать тестовое покрытие множества неисправностей, если тесты будут написаны на уровне объекта или подсистемы. К тому же, зачастую «белые» тесты легче написать и автоматизировать, чем «черные». В этой работе делается акцент на методах тестирования путем применения «прозрачного ящика», поскольку программист может использовать эти методы еще во время разработки своей программы.

Единственный способ выявить ошибки в программе – провести ее тестирование. Одним из самых важных видов тестирования с точки зрения разработчика является блочное, или поэлементное [**?**]. Поэлементные тесты представляют собой программы, которые проверяют работоспособность клас-

са или подсистемы. В идеальном варианте для каждой задачи низкого уровня должен существовать один или несколько тестов этого ранга.

Хорошо написанные тесты служат защитой во многих отношениях.

1) Они доказывают, что данная часть программы действительно работает должным образом. До тех пор, пока не будет получен код, который и в самом деле оправдывает существование тестируемого класса, его поведение можно считать неизвестным.
2) Поэлементные тесты первыми подают сигнал тревоги, если после недавнего изменения возник дефект.
3) Используемые как часть общего процесса разработки, они заставляют разработчика устранять проблемы с самого начала. Если какой-либо участок кода вообще не контролируется с помощью блочных тестов, то в случае возникновения проблемы ее источник можно смело искать в этом участке.
4) Поэлементные тесты позволяют испытать код до объединения с другим кодом.
5) Поэлементные тесты показывают пример применения созданного кода.

Увеличение количества написанных тестов влечет за собой расширение тестового покрытия кода и снижение вероятности невыявления ошибки.

Существуют разные методики написания поэлементных тестов [**?**]. Методология экстремального программирования (Extreme Programming methodology) предписывает своим сторонникам создавать тесты еще до написания кода. Теоретически предварительное написание тестов помогает четче сформулировать требования к компоненту и предложить систему показателей, которые могут быть использованы для определения момента завершения кода. Менее жесткий вариант состоит в проектировании тестов до кодирования, но в расчете на более позднюю их реализацию. В этом случае программист попрежнему вынужден четко понимать требования, предъявляемые к модулю, но не обязан писать код, использующий еще несуществующие классы.

Код теста во многом зависит от типа используемой тестовой оболочки. В данной работе используется оболочка **CppUnit** – это программный набор инструментов для модульного тестирования программ на языках программи-

рования C/C++ [**?**]. Система разрабатывается с 2000 г. и публикуется под GNU LGPL.

Оболочка запускает *тесты*, сгруппированные в *пакеты*. Результаты тестирования направляются в указанные фильтры, набор которых разнообразен: от простейшего подсчета пройденных либо не пройденных тестов до более продвинутых, поддерживающих XML-вывод, совместимый с дальнейшей интеграцией с системами отчетов.

Для того, чтобы понять, каким образом работает система тестирования `CppUnit`, рассмотрим, как спроектирован класс `MatrixTest`, основная задача которого – проведение блочного тестирования класса `Matrix`.

Объявление класса происходит в заголовочном файле (полный код класса приведен в приложении Б).

```
22 class MatrixTest : public CppUnit::TestFixture
23 {
```

Класс `MatrixTest` наследуется от класса `TestFixture`, который представляет собой логическую группу тестов.

Оболочка `CppUnit` выполняет группу тестов в виде некоторого набора (тестового комплекта), который должен содержать информацию о том, какие тесты подлежат выполнению, а какие нет (в отличие от класса `TestFixture`, который просто логически группирует тесты). Для этой цели `CppUnit` предоставляет класс `TestSuite`, который содержит в себе произвольный набор тестов. Обычная практика использования класса `TestSuite` состоит в следующем. В классы тестов (например такие, как `MatrixTest`), добавляется специальный статический метод.

```
1 public:
2   static CppUnit::Test* suite()
3   {
4     ...
5   }
```

После чего, посредством вызова этого метода, происходит добавление произвольного количества тестов в тестовый набор и запуск этого набора на выполнение. Как можно заметить, это довольно рутинная, повторяющаяся для каждого нового теста, задача. Поэтому `CppUnit` предоставляет набор

вспомогательных макросов, которые призваны упростить процесс создания наборов тестов.

Поскольку разрабатывается класс, представляющий собой описание математического объекта (матрицы) в программе, для проверки его работоспособности видится естественным проведение как минимум следующих тестов:

- тест сложения матриц;
- тест умножения матриц;
- тест транспонирования матриц;
- тесты производительности.

В заголовочном файле класса `MatrixTest` происходит объявление перечисленных тестов.

```
24     CPPUNIT_TEST_SUITE( MatrixTest );
25       CPPUNIT_TEST( testAddition );
26       CPPUNIT_TEST( testMultiplication );
27       CPPUNIT_TEST( testTransposition );
28       CPPUNIT_TEST( testAcceleration );
29       CPPUNIT_TEST( testMultithreadingTime );
30       CPPUNIT_TEST( testThreadsNumberTime );
31       CPPUNIT_TEST( testMatrixTypes );
```

Это один из примеров использования вспомогательных макросов. Такой код осуществляет автоматическое создание тестового набора `MatrixTest` и добавление тестов в набор для последующего выполнения.

Далее происходит объявление соответствующих методов класса.

```
33
34 public:
35
36   void testCrashDoubleType();
37   void testAddition();
38   void testMultiplication();
39   void testTransposition();
40   void testAcceleration();
41   void testMultithreadingTime();
```

Тест `testCrashDoubleType` не был добавлен в набор, потому что демонстрирует появление ошибки округления, возникающей при использовании типа `double` и проводит к систематическому непрохождению тестового на-

бора. Исходный код этого теста уже был проанализирован (см. листинг 1 и комментарии к нему).

Чтобы выполнить набор тестов и увидеть результаты, необходим *прогонщик тестов*. В оболочке `CppUnit` содержится несколько различных прогонщиков, которые действуют в различных средах (например, прогонщик `MFCRunner` предназначен для выполнения в программе, написанной с использованием библиотеки базовых классов Microsoft). Для текстовых сред используется прогонщик `TextRunner`. Ниже приводится код тела функции `main`, в котором происходит создание всех основных объектов тестового окружения.

```
14
15 int main( int argc, const char* argv[ ] )
16 {
17   CppUnit::TestResult controller;
18
19   CppUnit::TestResultCollector result;
20   controller.addListener( &result );
21
22   CppUnit::TextTestProgressListener progress;
23   controller.addListener( &progress );
24
25   CppUnit::TextUi::TestRunner runner;
26   runner.addTest( CppUnit::TestFactoryRegistry::getRegistry().makeTest() );
27
28   runner.run( controller, "" );
29
30   CppUnit::CompilerOutputter outputter( &result, std::cerr );
31   outputter.write();
32
33   return result.wasSuccessful() ? 0 : 1;
```

В строке 16 создается объект `controller` класса `TestResult`, представляющий собой статистику прохождения тестов. Другие вспомогательные классы будут подключаться именно к нему. В строке 18 создается сборщик статистики. В строке 21 – объект, следящий за ходом выполнения всего набора тестов. В строке 24 создается прогонщик тестов и запускается в строке 27. В строке 29 создается объект, отвечающий за печать результатов теста в совместимом с компилятором формате.

Рассмотрим структуру самих тестов, описанных в файле реализации. Для регистрации созданного набора тестов в системе используется такая строка.

```
14 CPPUNIT_TEST_SUITE_REGISTRATION( MatrixTest );
```

Для проверки утверждений и определения точного места, где они не выполняются, используется макрос `CPPUNIT_ASSERT`. Рассмотрим первый тест, где он применяется.

```
49 void MatrixTest::testAddition()
50 {
51   matrix_t A( this->matrixSize, this->matrixSize );
52   this->matrixRandomFiller.fill( A );
53
54   matrix_t B( this->matrixSize, this->matrixSize );
55   this->matrixRandomFiller.fill( B );
56
57   matrix_t C( this->matrixSize, this->matrixSize );
58   C = this->matrixSummarizer.summarize( A, B );
59
60   matrix_t D( B + A );
61
62   CPPUNIT_ASSERT( C == D );
63 }
```

Это *тест, проверяющий корректность сложения матриц*. В строках 51–55 с помощью уже известных объектов вспомогательных классов происходит создание и заполнение операндов, участвующих в тесте. В строке 58 осуществляется вычисление результата сложения двух операндов сторонним классом и сохранение этого результата в переменной `C`. В строке 60 сложение выполняет сам класс `Matrix`, сохраняя результат в переменную `D`. В строке 62 осуществляется формальная проверка на равенство двух независимо полученных результатов. В этой точке теста *ожидается*, что значения этих результатов совпадут. Если окажется, что это не так, `CppUnit` сообщит о провале прохождения теста и укажет на эту точку, сообщив имя файла и номер строки.

Тест матричного умножения во многом схож с первым тестом. Его исходный код приведен ниже.

```
67 void MatrixTest::testMultiplication()
68 {
69   matrix_t A( this->matrixSize, this->matrixSize );
```

```
70    this->matrixRandomFiller.fill( A );
71
72    matrix_t B( this->matrixSize, this->matrixSize );
73    this->matrixRandomFiller.fill( B );
74
75    matrix_t C( this->matrixSize, this->matrixSize );
76    C = this->matrixMultiplier.multiply( A, B );
77
78    matrix_t D( A * B );
79
80    CPPUNIT_ASSERT( C == D );
81
82    matrix_t E( A );
83    element_t value = 2;
84    this->matrixMultiplier.multiply( value, E );
85
86    matrix_t F( A * value );
87
88    CPPUNIT_ASSERT( A == E );
89  }
```

Единственное отличие состоит в дополнительной проверке в строках 82–88 работы операции умножения матрицы на число.

Тест транспонирования необходим для контроля правильности выполнения соответствующей матричной операции.

```
93  void MatrixTest::testTransposition()
94  {
95    matrix_t matrix( this->matrixSize, 2 * this->matrixSize );
96    this->matrixRandomFiller.fill( matrix );
97
98    matrix_t matrix_copy = matrix;
99
100   matrix_t matrix_transp( matrix.get_columns(), matrix.get_rows() );
101   this->matrixTransposer.transpose( matrix_transp, matrix );
102
103   matrix.transpose();
104
105   CPPUNIT_ASSERT( matrix == matrix_transp );
106
107   const std::size_t row = 4;
108   const std::size_t column = 5;
109
110   CPPUNIT_ASSERT( matrix_copy( row, column ) == matrix( column, row ) );
111
```

```
112     matrix.transpose();
113
114     CPPUNIT_ASSERT( matrix == matrix_copy );
115     CPPUNIT_ASSERT( matrix( row, column ) == matrix_copy( row, column ) );
116 }
```

Рассмотрим тест подробнее. В строке 95 создается случайно заполненная матрица, и ее первоначальное значение в строке 98 дублируется. В строке 101 сторонний класс транспонирует исходную матрицу, сохраняя результат в переменной `matrix_transp`. В строке 103 исходную матрицу транспонирует сам класс, после чего, в строке 105, результат операции сразу проверяется. В строке 110 производится проверка доступа по индексу. Смысл этой операции в том, чтобы убедиться в свойстве зеркальности операции транспонирования, а именно что (i, j)-й элемент исходной матрицы соответствует (j, i)-му элементу транспонированной. В строке 112 матрица транспонируется обратно, и в строке 114 проверяется на соответствие своему первоначальному значению. В строке 115 осуществляется еще одна проверка индексного доступа.

Тестовый набор содержит дополнительно тесты производительности, осуществляющие замеры времени работы разработанного класса. Их исходный код приведен в приложении Б .

Выводы по главе два

В главе дается подробное объяснение почему для распараллеливания лучше всего подходит модель Fork-Join параллельного запуска участков кода.

Рассмотрены все блочные тесты, участвующие в проверке корректности работы разрабатываемого класса `Matrix`. Их регулярное прохождение будет доказывать правильность работы каждого участка кода. А в случае возникновения ошибки оболочка `CppUnit` позволит быстро локализовать и устранить проблему.

Заключение

В работе представлена параллельная реализация алгоритма метода эллипсоидов.

В работе выполнены следующие **задачи**:

- операций классического алгоритма метода эллипсоидов исследованы на вычислительную сложность;
- разработана программная реализация алгоритма с распараллеливанием наиболее длительных по времени операций;
- обеспечена поддержка арифметики расширенной и произвольной точности;
- разработанный код проверен и протестирован.

После выполнения вычислительных экспериментов можно сделать следующие вывод: для задач оптимизации большой размерности разработанная реализация выполняется быстрее однопоточного алгоритма и имеет ускорение, большее 1.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

(обязательное)

ИСХОДНЫЙ КОД КЛАССА MATRIX

Листинг А.1 — Файл «Matrix.h»

```
#ifndef MATRIX_H
#define MATRIX_H

#include <assert.h>
#include <valarray>

#include "ParallelHandler.h"



template< typename T >
class Matrix
{
private:

  typedef T value_t;
  typedef Matrix< value_t > my_t;
  typedef std::size_t index_t;



public:

  // Construct a rows-by-columns Matrix filled with the given value
  Matrix( index_t rows, index_t columns, const value_t& value = value_t( 0 ) )
    : values( value, rows * columns )
    , rows( rows )
    , columns( columns )
    , isTransposed( false )
  {
    assert( rows > 0 );
    assert( columns > 0 );
  }



  // Construct a copy of other Matrix object
```

```
38  Matrix( const my_t& rhs )
39    : values( rhs.values )
40    , rows( rhs.rows )
41    , columns( rhs.columns )
42    , isTransposed( rhs.isTransposed )
43  {
44
45  }
46
47
48
49  // Construct a copy using resources of temporary Matrix object
50  Matrix( my_t&& rhs )
51    : values( std::move( rhs.values ) )
52    , rows( rhs.rows )
53    , columns( rhs.columns )
54    , isTransposed( rhs.isTransposed )
55  {
56
57  }
58
59
60
61  // Retrieve the reference
62  value_t& operator() ( const index_t& row, const index_t& column )
63  {
64    assert( row < this->get_rows() );
65    assert( column < this->get_columns() );
66
67    return this->get( row, column );
68  }
69
70
71
72  // Retrieve the const reference
73  const value_t& operator() ( const index_t& row, const index_t& column ) const
74  {
75    assert( row < this->get_rows() );
76    assert( column < this->get_columns() );
77
78    return this->get( row, column );
79  }
80
81
82
83  // Make the matrix equal to other Matrix object
```

```
84  my_t& operator= ( const my_t& rhs )
85  {
86    if( this == &rhs ) {
87      return *this;
88    }
89
90    this->values = rhs.values;
91    this->rows = rhs.rows;
92    this->columns = rhs.columns;
93    this->isTransposed = rhs.isTransposed;
94
95    return *this;
96  }
97
98
99
100 // Make the matrix the owner of the resources of the temporary Matrix object
101 my_t& operator= ( my_t&& rhs )
102 {
103   if (this == &rhs) {
104     return *this;
105   }
106
107   this->values = std::move( rhs.values );
108   this->rows = rhs.rows;
109   this->columns = rhs.columns;
110   this->isTransposed = rhs.isTransposed;
111
112   return *this;
113 }
114
115
116
117 // Retrieve the Matrix each element of which is the sum of the corresponding
118 // values of this and rhs Matrix objects
119 my_t operator+ ( const my_t& rhs ) const
120 {
121   assert( this->get_rows() == rhs.get_rows() );
122   assert( this->get_columns() == rhs.get_columns() );
123
124   my_t result( this->get_rows(), this->get_columns() );
125
126   auto summarize = [ this, &rhs, &result ]( const index_t& iRowStart,
127     const index_t& iRowEnd ) -> void
128   {
129     for( index_t iRow = iRowStart; iRow < iRowEnd; ++iRow ) {
```

```
130         for( index_t jColumn = 0; jColumn < this->get_columns(); ++jColumn ) {
131           result.get( iRow, jColumn ) = (
132             this->get( iRow, jColumn ) + rhs.get( iRow, jColumn )
133           );
134         }
135       }
136     };
137
138     const index_t first = 0;
139     const index_t last = this->get_rows();
140     my_t::getParallelHandler()->loop_for( first, last, summarize );
141
142     return result;
143   }
144
145
146
147   // Retrieve the Matrix each element of which is the difference of the correspondin
148   // values of this and rhs Matrix objects
149   my_t operator- ( const my_t& rhs ) const
150   {
151     assert( this->get_rows() == rhs.get_rows() );
152     assert( this->get_columns() == rhs.get_columns() );
153
154     my_t result( this->get_rows(), this->get_columns() );
155
156     auto subtract = [ this, &rhs, &result ]( const index_t& iRowStart,
157       const index_t& iRowEnd ) -> void
158     {
159       for( index_t iRow = iRowStart; iRow < iRowEnd; ++iRow ) {
160         for( index_t jColumn = 0; jColumn < this->get_columns(); ++jColumn ) {
161           result.get( iRow, jColumn ) = (
162             this->get( iRow, jColumn ) - rhs.get( iRow, jColumn )
163           );
164         }
165       }
166     };
167
168     const index_t first = 0;
169     const index_t last = this->get_rows();
170     my_t::getParallelHandler()->loop_for( first, last, subtract );
171
172     return result;
173   }
174
175
```

```
176
177   // Retrieve the Matrix which is equal to result of matrix multiplication
178   // of this and rhs Matrix objects
179   my_t operator* ( const my_t& rhs ) const
180   {
181     assert( this->get_columns() == rhs.get_rows() );
182
183     my_t result( this->get_rows(), rhs.get_columns() );
184
185     auto multiply = [ this, &rhs, &result ]( const index_t& iRowStart,
186       const index_t& iRowEnd ) -> void
187     {
188       for( index_t iRow = iRowStart; iRow < iRowEnd; ++iRow ) {
189         for( index_t jColumn = 0; jColumn < rhs.get_columns(); ++jColumn ) {
190           for( index_t sIndex = 0; sIndex < this->get_columns(); ++sIndex ) {
191             result.get( iRow, jColumn ) += (
192               this->get( iRow, sIndex ) * rhs.get( sIndex, jColumn )
193             );
194           }
195         }
196       }
197     };
198
199     const index_t first = 0;
200     const index_t last = this->get_rows();
201     my_t::getParallelHandler()->loop_for( first, last, multiply );
202
203     return result;
204   }
205
206
207
208   // Multiply each element of the Matrix by value
209   my_t operator* ( const T& value )
210   {
211     my_t result( *this );
212
213     auto multiply = [ &result, &value ]( const index_t& iRowStart,
214       const index_t& iRowEnd ) -> void
215     {
216       for( index_t iRow = iRowStart; iRow < iRowEnd; ++iRow ) {
217         for( index_t jColumn = 0; jColumn < result.get_columns(); ++jColumn ) {
218           result.get( iRow, jColumn ) *= value;
219         }
220       }
221     };
```

```
222
223       const index_t first = 0;
224       const index_t last = result.get_rows();
225       my_t::getParallelHandler()->loop_for( first, last, multiply );
226
227       return result;
228     }
229
230
231
232     // Check if each element of the matrix is equal to the corresponding
233     // element of the other Matrix object
234     bool operator== ( const my_t& rhs ) const
235     {
236       if ( &rhs == this ) {
237         return true;
238       }
239
240       bool rowsMismatch = ( this->get_rows() != rhs.get_rows() );
241       bool columnsMismatch = ( this->get_columns() != rhs.get_columns() );
242
243       if ( rowsMismatch || columnsMismatch ) {
244         return false;
245       }
246
247       std::valarray< bool > comp = ( this->values == rhs.values );
248       bool isEqual = ( comp.min() == true );
249
250       return isEqual;
251     }
252
253
254
255     // Check if the matrix is not equal to the other Matrix object
256     bool operator!= ( const my_t& rhs ) const
257     {
258       return !( *this == rhs );
259     }
260
261
262
263     // Swap rows and columns
264     my_t& transpose()
265     {
266       this->isTransposed = !this->isTransposed;
267
```

```
268     return *this;
269   }
270
271
272
273   // Retrieve the number of rows of the matrix
274   std::size_t get_rows() const
275   {
276     return ( this->isTransposed ? this->columns : this->rows );
277   }
278
279
280
281   // Retrieve the number of columns of the matrix
282   std::size_t get_columns() const
283   {
284     return ( this->isTransposed ? this->rows : this->columns );
285   }
286
287
288
289   // Retrieve an instance of parallel execution controller
290   static ParallelHandler* getParallelHandler()
291   {
292     static ParallelHandler parallelHandler;
293
294     return &parallelHandler;
295   }
296
297
298
299   class Type
300   {
301   public:
302
303     static Matrix< T > Identity( index_t size )
304     {
305       Matrix< T > identityMatrix( size, size );
306
307       for( index_t index = 0; index < size; ++index ) {
308         identityMatrix( index, index ) = value_t( 1 );
309       }
310
311       return identityMatrix;
312     }
313   };
```

```
314
315
316
317 private:
318
319   // Retrieve the reference by index
320   value_t& get( index_t row, index_t column )
321   {
322     if( this->isTransposed ) {
323       std::swap( row, column );
324     }
325
326     index_t index = ( row * this->columns + column );
327
328     return this->values[ index ];
329   }
330
331
332
333   // Retrieve the const reference by index
334   const value_t& get( index_t row, index_t column ) const
335   {
336     if( this->isTransposed ) {
337       std::swap( row, column );
338     }
339
340     index_t index = ( row * this->columns + column );
341
342     return this->values[ index ];
343   }
344
345
346
347   std::valarray< value_t > values;
348   index_t rows;
349   index_t columns;
350
351   bool isTransposed;
352 };
353
354
355
356 #endif // MATRIX_H
```

ПРИЛОЖЕНИЕ Б

(обязательное)

ИСХОДНЫЙ КОД КЛАССА MATRIXTEST

Листинг Б.1 — Файл «MatrixTest.h»

```
1  #ifndef MATRIX_TEST
2  #define MATRIX_TEST
3
4  #include <memory>
5  #include <gmpxx.h>
6
7  #include <cppunit/extensions/HelperMacros.h>
8  #include <cppunit/TestFixture.h>
9  #include <cppunit/Test.h>
10
11 #include "Matrix.h"
12
13 #include "HelperClasses/MatrixMultiplier.h"
14 #include "HelperClasses/MatrixRandomFiller.h"
15 #include "HelperClasses/MatrixSummarizer.h"
16 #include "HelperClasses/MatrixTransposer.h"
17 #include "HelperClasses/Statistics.h"
18 #include "HelperClasses/TimeMeasurer.h"
19
20
21
22 class MatrixTest : public CppUnit::TestFixture
23 {
24     CPPUNIT_TEST_SUITE( MatrixTest );
25       CPPUNIT_TEST( testAddition );
26       CPPUNIT_TEST( testMultiplication );
27       CPPUNIT_TEST( testTransposition );
28       CPPUNIT_TEST( testAcceleration );
29       CPPUNIT_TEST( testMultithreadingTime );
30       CPPUNIT_TEST( testThreadsNumberTime );
31       CPPUNIT_TEST( testMatrixTypes );
32     CPPUNIT_TEST_SUITE_END();
33
34 public:
35
36   void testCrashDoubleType();
37   void testAddition();
```

```
38    void testMultiplication();
39    void testTransposition();
40    void testAcceleration();
41    void testMultithreadingTime();
42    void testThreadsNumberTime();
43    void testMatrixTypes();
44
45  private:
46
47    template< typename Function >
48    double calculateAverageTime( Function&& function ) {
49      TimeMeasurer timeMeasurer;
50      std::valarray< double > durations( this->nInnerLoopIterations );
51      for( std::size_t iDuration = 0; iDuration < durations.size(); ++iDuration ) {
52        timeMeasurer.start();
53        function();
54        timeMeasurer.end();
55        durations[ iDuration ] = timeMeasurer.getDurationInSeconds();
56      }
57
58      return durations.sum() / durations.size();
59    }
60
61
62
63    typedef mpz_class element_t;
64    typedef Matrix< element_t > matrix_t;
65
66    std::size_t matrixSize = 50;
67    std::size_t initialSize = 10;
68    std::size_t sizeStep = 10;
69    std::size_t nIterations = 0;
70    std::size_t nInnerLoopIterations = 1;
71
72    Statistics statistics;
73
74    MatrixRandomFiller matrixRandomFiller;
75    MatrixSummarizer matrixSummarizer;
76    MatrixMultiplier matrixMultiplier;
77    MatrixTransposer matrixTransposer;
78  };
79
80
81
82  #endif // MATRIX_TEST
```

Листинг Б.2 — Файл «MatrixTest.cpp»

```
1  #include "MatrixTest.h"
2
3  #include <valarray>
4
5  #include <cppunit/TestCaller.h>
6  #include <cppunit/TestSuite.h>
7
8  #include "HelperClasses/Logger.h"
9  #include "HelperClasses/MatrixPrinter.h"
10 #include "HelperClasses/Statistics.h"
11
12
13
14 CPPUNIT_TEST_SUITE_REGISTRATION( MatrixTest );
15
16
17
18 void MatrixTest::testCrashDoubleType()
19 {
20   MatrixRandomFiller filler;
21   MatrixMultiplier multiplier;
22   MatrixPrinter printer( &std::cout );
23   printer.setPrecision( 20 );
24
25   std::size_t nRows = 2;
26   std::size_t nColumns = 2;
27
28   Matrix< double > matrix_a( nRows, nColumns );
29   Matrix< double > matrix_b( nRows, nColumns );
30
31   filler.fill( matrix_a );
32   filler.fill( matrix_b );
33
34   printer.print( matrix_a );
35   printer.print( matrix_b );
36
37   Matrix< double > matrix_c( nRows, nColumns );
38   matrix_c = multiplier.multiply( matrix_a, matrix_b );
39
40   Matrix< double > matrix_d( nRows, nColumns );
41   matrix_d = matrix_a * matrix_b;
42
43   printer.print( matrix_c );
```

```
44    printer.print( matrix_d );
45  }
46
47
48
49  void MatrixTest::testAddition()
50  {
51    matrix_t A( this->matrixSize, this->matrixSize );
52    this->matrixRandomFiller.fill( A );
53
54    matrix_t B( this->matrixSize, this->matrixSize );
55    this->matrixRandomFiller.fill( B );
56
57    matrix_t C( this->matrixSize, this->matrixSize );
58    C = this->matrixSummarizer.summarize( A, B );
59
60    matrix_t D( B + A );
61
62    CPPUNIT_ASSERT( C == D );
63  }
64
65
66
67  void MatrixTest::testMultiplication()
68  {
69    matrix_t A( this->matrixSize, this->matrixSize );
70    this->matrixRandomFiller.fill( A );
71
72    matrix_t B( this->matrixSize, this->matrixSize );
73    this->matrixRandomFiller.fill( B );
74
75    matrix_t C( this->matrixSize, this->matrixSize );
76    C = this->matrixMultiplier.multiply( A, B );
77
78    matrix_t D( A * B );
79
80    CPPUNIT_ASSERT( C == D );
81
82    matrix_t E( A );
83    element_t value = 2;
84    this->matrixMultiplier.multiply( value, E );
85
86    matrix_t F( A * value );
87
88    CPPUNIT_ASSERT( A == E );
89  }
```

```
90
91
92
93  void MatrixTest::testTransposition()
94  {
95    matrix_t matrix( this->matrixSize, 2 * this->matrixSize );
96    this->matrixRandomFiller.fill( matrix );
97
98    matrix_t matrix_copy = matrix;
99
100   matrix_t matrix_transp( matrix.get_columns(), matrix.get_rows() );
101   this->matrixTransposer.transpose( matrix_transp, matrix );
102
103   matrix.transpose();
104
105   CPPUNIT_ASSERT( matrix == matrix_transp );
106
107   const std::size_t row = 4;
108   const std::size_t column = 5;
109
110   CPPUNIT_ASSERT( matrix_copy( row, column ) == matrix( column, row ) );
111
112   matrix.transpose();
113
114   CPPUNIT_ASSERT( matrix == matrix_copy );
115   CPPUNIT_ASSERT( matrix( row, column ) == matrix_copy( row, column ) );
116 }
117
118
119
120 void MatrixTest::testAcceleration()
121 {
122   std::size_t size = this->initialSize;
123   for( std::size_t iteration = 0; iteration < nIterations; ++iteration ) {
124     matrix_t lhs( size, size );
125     this->matrixRandomFiller.fill( lhs );
126
127     matrix_t rhs( size, size );
128     this->matrixRandomFiller.fill( rhs );
129
130     matrix_t resultExternal( size, size );
131     matrix_t resultInternal( size, size );
132
133     auto externalSummarizing = [ & ]() {
134       resultExternal = this->matrixSummarizer.summarize( lhs, rhs );
135     };
```

```
136
137       auto internalSummarizing = [ & ]() {
138         resultInternal = ( lhs + rhs );
139       };
140
141       auto externalMultiplication = [ & ]() {
142         resultExternal = this->matrixMultiplier.multiply( lhs, rhs );
143       };
144
145       auto internalMultiplication = [ & ]() {
146         resultInternal = ( lhs * rhs );
147       };
148
149       this->statistics( size, Statistics::Addition ).setSingleThreadTime(
150         this->calculateAverageTime( externalSummarizing )
151       );
152
153       this->statistics( size, Statistics::Addition ).setMultyThreadTime(
154         this->calculateAverageTime( internalSummarizing )
155       );
156
157       this->statistics( size, Statistics::Multiplication ).setSingleThreadTime(
158         this->calculateAverageTime( externalMultiplication )
159       );
160
161       this->statistics( size, Statistics::Multiplication ).setMultyThreadTime(
162         this->calculateAverageTime( internalMultiplication )
163       );
164
165       size = this->initialSize * (this->sizeStep + 10 * iteration);
166     }
167
168     const std::string fileName = "Acceleration statistics";
169     this->statistics.save( fileName );
170 }
171
172
173
174 void MatrixTest::testMultithreadingTime()
175 {
176     this->nInnerLoopIterations = 1;
177     this->initialSize = 500;
178     this->sizeStep = 500;
179     this->nIterations = 0;
180
181     std::size_t size = this->initialSize;
```

```
182   for( std::size_t iteration = 0; iteration < this->nIterations; ++iteration ) {
183     matrix_t lhs( size, size );
184     this->matrixRandomFiller.fill( lhs );
185
186     matrix_t rhs( size, size );
187     this->matrixRandomFiller.fill( rhs );
188
189     matrix_t resultInternal( size, size );
190
191     auto internalSummarizing = [ & ]() {
192       resultInternal = ( lhs + rhs );
193     };
194
195     auto internalMultiplication = [ & ]() {
196       resultInternal = ( lhs * rhs );
197     };
198
199     this->statistics( size, Statistics::Addition ).setMultyThreadTime(
200       this->calculateAverageTime( internalSummarizing )
201     );
202
203     this->statistics( size, Statistics::Multiplication ).setMultyThreadTime(
204       this->calculateAverageTime( internalMultiplication )
205     );
206
207     size += this->sizeStep;
208   }
209
210   const std::string fileName = "Multithreading statistics";
211   this->statistics.save( fileName );
212 }
213
214
215
216 void MatrixTest::testThreadsNumberTime()
217 {
218   const std::size_t matrixSize = 500;
219   this->nInnerLoopIterations = 0;
220
221   matrix_t lhs( matrixSize, matrixSize );
222   this->matrixRandomFiller.fill( lhs );
223
224   matrix_t rhs( matrixSize, matrixSize );
225   this->matrixRandomFiller.fill( rhs );
226
227   matrix_t result( matrixSize, matrixSize );
```

```
228
229   auto internalMultiplication = [ & ]() {
230     result = ( lhs * rhs );
231   };
232
233   double time = this->calculateAverageTime( internalMultiplication );
234
235   const std::string fileName = "Time (using 2 threads)";
236   std::ofstream fstream( fileName );
237
238   fstream << time;
239 }
240
241
242
243 void MatrixTest::testMatrixTypes()
244 {
245   Matrix< int > E( 3, 3 );
246   E( 0, 0 ) = 1;
247   E( 1, 1 ) = 1;
248   E( 2, 2 ) = 1;
249   CPPUNIT_ASSERT( E == Matrix< int >::Type::Identity( 3 ) );
250 }
```